Pedro Filipe Nunes Durães

Integration of browser-to-browser architectures with third party legacy cloud storage

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Nuno Manuel Ribeiro Preguiça,
Professor Associado,
Universidade Nova de Lisboa

Co-orientador: João Leitão,
Professor Assistente,
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2016

Integration of browser-to-browser architectures with third party legacy cloud storage

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2]. [1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

To my family and friends.

Acknowledgements

I would firstly like to thank my adviser professors, Nuno Preguiça and João Leitão for helping and pushing me ahead throughout this thesis, to Albert Linde for helping me with Legion related questions and to the other msc students with related thesis that I was able to share information with. I would also like to thank the SyncFree consortium members that made their time to help me.

Besides the ones who directly helped me with the work, I would also like to think my colleagues that were either also doing their thesis or having classes for motivating me along the way. Also a thanks to my hometown friends for always cheering me up, and last but not least, a big thanks to my parents that made possible for me to take part in this course. I do apologize if at this time I forgot to mention someone.

Abstract

An increasing number of web applications run totally or partially in the client machines - from collaborative editing tools to multi-user games. Avoiding to continuously contact the server allows to reduce latency between clients and to minimize the load on the centralized component. Novel implementation techniques, such as WebRTC, allows to address network problems that previously prevented these systems to be deployed in practice. Legion is a newly developed framework that exploits these mechanisms, allowing client web applications to replicate data from servers, and synchronize these replicas directly among them.

This work aims at extending the current Legion framework with the integration of an additional legacy storage system that can be used to support web applications, Antidote. We study the best way to support Legion's data model into Antidote, we design a synchronization mechanism, and finally we measure the performance cost of such an integration.

Keywords: distributed storage systems; CRDT; Legion; Antidote.

Resumo

Um crescente número de aplicações *web* é executado totalmente ou parcialmente nas máquinas do cliente - desde ferramentas de edição colaborativa até jogos de vários utilizadores. Ao cortar a execução de operações no servidor é possível reduzir a latência entre clientes e minimizar a carga no servidor. Novas tecnologias, como WebRTC, permitem atacar problemas que anteriormente afectavam a implementação destes sistemas num ambiente de produção. O Legion, é uma *framework* recém desenvolvida que explora estes mecanismos, permitindo que aplicações *web* do lado do cliente consigam fazer replicação de dados e sincronização directa entre elas.

Neste trabalho pretende-se estender a *framework* Legion através da integração com um sistema de armazenamento já existente que pode ser utilizado para suportar aplicações *Web*, o Antidote. Estudamos o melhor método para suportar o modelo de dados do Legion no Antidote, desenhamos o mecanismo de sincronização, e finalmente medimos o custo no desempenho de tal integração.

Palavras-chave: sistemas de armazenamento distribuido; CRDT; Legion; Antidote.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Web applications have taken a central role in access to remote services. Traditionally, clients would only act as an end interface of the data from/to the server, not keeping data nor making computation, with the exception of computation related to data presentation. With the grow of processing power and storage capabilities of commodity hardware, such as PCs, Laptops, SmartPhones and Tablets, there has been a tendency to move some of the computation and data to these devices, instead of always relying on the servers.

Peer-to-peer technologies have been around for long, but it has not been used in web applications due to the difficulty of having one client communicating with other clients. When communicating directly among clients, there can be benefits in terms of (i) latency, since devices can be closer to each other than to a server; (ii) scaling, work can be partitioned between peers; (iii) availability, because a service doesn't need to stop if the central server is temporarily down. Recent technology developments in this area, such as WebRTC, STUN and TURN, allowed browser-to-browser communication with no need for native applications or browser plugins.

As an example of this, Legion is a newly created framework which explores these technologies and allows client web applications to replicate data from servers, and synchronize these replicas directly among them. Browser-to-browser communication is useful in web applications that exhibit frequent interchange of information or sharing of contents between clients, such as collaborative editing.

Although replicating data between web clients is a promising approach, personal devices can be unstable compared to the use of a centralized component server because they join and leave a network very often, so data persistency cannot be fully delegated to these devices. There is a need store data periodically in a more stable storage system.

1.2 Motivating Problem and Solution

The processing power and the amount of storage of user devices allow current systems to transfer part of the work and data from a central server (or servers) to these end-point devices. There has been an effort to build tools that facilitate programmers to develop software that takes advantage of direct communication between clients.

In this scenario, communication between browsers is possible with minimal effort from both developer and user, thanks to frameworks like WebRTC that enable direct real time communication between browsers. With WebRTC it is possible to stream data, audio, or video between browsers with no need for plugins or extensions.

WebRTC made possible frameworks like Legion. Legion allows client web applications to replicate data from servers, and synchronize these replicas directly among them. This can have major impact in areas like collaborative editing, where current approaches like Google Docs always use a central server do mediate communication between clients. Legion offers the same API as Google Drive Realtime, thus allowing to easily port existing applications to the new framework.

However, there are some disadvantages in direct browser-to-browser communication. User devices are unstable when compared to a server on a data center, as they can join and leave the system frequently. This makes imperative to include in the system a centralized component where data durability is guaranteed. Part of the goal of this thesis is to study how to integrate a legacy distributed storage system with the Legion framework, giving this framework more robustness when it comes to persistence of data and allowing clients that don't support WebRTC to use the framework as an old fashion centralized server approach.

This integration will have the following main challenges. First, the incompatibility between the Legion framework data model and the storage systems data models. To address this challenge, it will be necessary to both extend the Legion data model and to create a mapping between the two models. Second, the need to keep data synchronized between the central storage and the Legion framework, considering that replicas can be modified concurrently. This encompasses the following problems: (i) identifying the updates that have been executed in the central server and in Legion; (ii) propagate the updates across systems efficiently. Besides this, and based on the study of distributed storage systems techniques, an important topic is consistency guarantees in the data propagation. Most well known storage systems offer some sort of consistency policy, whether this is atomic transactions, eventual consistency, or causal consistency. Currently, Legion support causal consistency for each object, but not across objects. In this work, we will need to settle the best relation between Legion's and the storage systems' consistency model. This will help peer-to-peer application developers to have an easier time reasoning about data propagation between nodes.

1.3 Expected Contributions

The planned work for this thesis will be based on the understanding of Legion and the study of well known legacy distributed storage systems. The expected contributions of this work will be:

- Contribute to Antidote's development by extending the support to a more language agnostic client interface.
- Develop a javascript client for Antidote.
- Extend Legion to support the integration with an existing storage system, such as Antidote.

1.4 Document Structure

The remainder of this document is organized as follows:

Chapter 2 describes the related work. Existing work is explored in the areas of communication technologies, peer-to-peer systems and distributed storage systems.

Chapter 3 discusses the proposed design, mentioning the options taken at each step and depicts the system architecture on the which the implementation will have its basis on.

Chapter 4 describes the process of the system implementation, including the synchronization algorithm and implementation details.

Chapter 6 concludes this document by summarizing the thesis, as well as discussing pointers for future work.

Chapter 2

Related Work

In this chapter, will be presented various aspect that will help with the work to be developed in this thesis. The following sections cover in particular:

- In section 2.1, an overall study of the WebRTC technology is presented.
- In section 2.2, storage systems mechanisms and examples are depicted, as this is one of the most important topics for this thesis.
- In section 2.3, peer-to-peer system mechanisms are explored, as they are widely used, even in storage systems.

2.1 WebRTC

WebRTC[4] is a framework for the web that enables Real Time Communications among browsers. Before the availability of WebRTC, real time communication was either done via native applications or plugins, which demanded large downloads and/or a great effort for both developers and users to install and keep updated. These disadvantages would make web-based applications that resort to direct communication among clients not viable for both operators and users alike. With WebRTC, the final user can have a much better experience on its browser and the developers can benefit from a structured and easy to use API to develop Web applications.

WebRTC includes three main components: audio, video and network. In the network component are mechanisms that to deal with network related practical issues. Also included are components for facilitating the establishment of peer-to-peer connections using ICE / STUN / Turn / RTP-over-TCP as well as support for proxies.

Although WebRTC was design to be used in Peer-to-Peer contexts, it relies on a centralized component for particular interactions:

- Before any connection can be made, WebRTC clients (peers) need to exchange network information (signaling protocol).

- For streaming media connections, peers must also exchange data about the media contents being exchanged, such as video encoding and resolution

- Additionally, as clients often reside behind NAT[1] gateways and firewalls, these may have to be traversed using STUN (Session Traversal Utilities for NAT) or TURN (Traversal Using Relays around NAT) servers.

2.1.1 Signaling

Signaling[31] is the process of coordinating communication. In order for a WebRTC application to set up a "call", its clients need to exchange the following information information: (i) session control messages used to open or close communication; (ii) error messages (iii) media metadata such as codecs and codec settings, bandwidth and media types (iv) key data, used to establish secure connections (v) network data, such as a host's IP address and port as seen by the outside world

This signaling process needs a way for clients to pass messages back and forth. This mechanism is not implemented by the WebRTC APIs, it must be implemented by the application developer. This implementation can be a simple messaging system that uses, for example, a central server.

2.1.2 STUN

NATs provide a device with an IP address for use within a private local network, but this address can't be used externally. Without a public address, there's no way for WebRTC peers to communicate. To get around this problem WebRTC uses STUN.

STUN severs have the task of checking the IP address and port of an incoming request from an application behind a NAT, and send that address back as response. This makes possible for an application to find its IP address and port from a public perspective.

2.1.3 TURN

TURN is used to relay audio/video/data streaming between peers. When direct communication via UDP and TCP fails, TURN servers can be used as a fallback. Unlike STUN servers, that do a simple task and do not require much bandwidth, TURN servers are heavier to support, as they relay data between peers.

[1] Network address translation.

2.2 Storage Systems

Storage is a fundamental aspect for the majority of applications and web applications in particular. Current web applications demand a set of characteristics that fit with their goals, whether this is strong consistency, high availability, global geo-replication of data, or high performance read and write operations.

2.2.1 Data Models

A data model that a storage system uses as internal representation will influence the outcome performance of the system at many levels: (i) querying speed, as the organization of data will influence how fast can the system find it; (ii) scalability, because different amounts of data requires different data structures; (iii) querying functionalities/operations, because how the data is stored will influence the different ways of searching and operating with that information.

2.2.1.1 Relational Model

The relational model[9] is vastly used in traditional databases. It represents data with a collection of relations. Each relation is a table that stores a collection of entities, where each row is a single entity and each column an attribute. Every table has one or more attributes that together make a unique key. This key allows efficient queries by indexing the table.

The relational model allows powerful queries by relying on relational algebra. Commonly used operations between tables include Select, Join, Intersect, Union, Difference and Product.

2.2.1.2 Key-Value Model

Key-Value stores use keys associated with values, making a collection of pairs. Storing and retrieving information from a key-value database can be very efficient, because it can rely on a dictionary to keep track of all the keys, using a hash function to access a position of the dictionary. Although more efficient in certain conditions, this model doesn't support such powerful querying as the relational model.

The key-value model has been getting more traction recently, as multiple databases and storage systems use it due to its scalability potential.

2.2.1.3 Bigtable Model

Bigtable[8] uses tables as its data model, but not in a relational way. A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map. Data is organized into three dimensions: rows, columns and timestamps. In order to access a specific storage cell, one needs to reference the row key, column key and timestamp. Columns can be

grouped into column families, which form the unit of access control. Table updates are executed at a transactional row level.

This approach allows the system to scale, while still offering some control over the data. System scalability can be guaranteed by allowing concurrent access to different rows, since Bigtable offers transactions per row.

2.2.2 Distributed Storage Systems Techniques

Each storage system has to address multiple aspects related with the storage and manipulation of data. These aspects include data partitioning when considering systems that are materialized by multiple machines, replication to ensure data availability and good performance, data versioning, membership control, failure handling, scalability, and overall system performance.

Each storage system tackles these aspects using different solutions to best fit a specific working environment or particular needs of the system operating on top of them.

2.2.2.1 Consistency Guarantees

There is a well known trade-of between performance and consistency in replicated systems, and in particular in geo-replicated systems[12]. In a nutshell, the greater the consistency guarantees offered by a system, the less performant that system becomes. The main consistency models are presented below:

Strong Consistency With Strong Consistency models strive to have all updates serialized in the same order across all replicas. While this can benefit the programmer by making it easier to reason about the state of a system, offering such guarantees makes the system less performant, and can even halt the system, when replicas cannot be contacted, for instance when there is a network partition.

Snapshot Isolation Snapshot isolation[19] guarantees that all reads made in a transaction will see a consistent snapshot of the system taken in the start of that transaction. This transaction will be successful if the updated values have not been changed since the beginning of the transaction, when the snapshot was taken.

Eventual Consistency Eventual consistency[3] guarantees that all updates will reach every replica eventually and that the state of all replicas will eventually converge. This usually means that update operations (i.e. operations that modify the state of the system) can usually be executed locally at one replica without resorting to coordination(in the critical path of operation execution). This implies that operations return their replies to clients before the operation becomes known by all replicas in the system.

This allows clients that issue read operations, to observe old, or divergent versions (from other clients reading the same content) of the data depending on which replica processes the client read operation.

Causal Consistency Causal consistency is a consistency model, whose guarantees lie somewhere in between strong consistency and eventual consistency. While it allows clients to observe divergent values, contrary to eventual consistency, it guarantees that clients observe values that respect causal relationships between them[2, 18]. A causal dependency is formed, for example, when a node performs a read followed later by a write, even on a different variable, the first operation is said to be causally ordered before the second, because the value stored by the write may have been dependent upon the result of the read.

In practice, respecting the causal relationship on these objects implies that a client should not be able to observe the effect of the write operation discussed above without being able to observe the same (as a latter version) value read by the client that issued that write.

2.2.2.2 Partitioning

In distributed storage systems, multiple nodes materialize the state of the database. This is done to achieve both load balancing and scalability (due to fault tolerance it is also crucial to replicate the same data across multiple nodes). Bellow are presented the main methods to do this:

Consistent Hashing Consistent Hashing[15] is about using a hash function to map each data item to a location(node) in the system. The range of values of the hash function can form a ring by overlapping the highest hash value with the lowest value. Each node in the system is assigned a random value within this range. Searching an entry in such system can be very efficient, because it only requires to apply the hash function to find the item location. Keeping this structure has maintenance costs, because the system needs to update the active nodes as they leave or join.

Directory Based lookups [21] Directory-based approaches can place replicas in arbitrary locations but require a lookup to the directory to find the replica location. This may involve one or more round-trips in the network. This technique has less maintenance costs then consistent hashing, as node joins and leaves are not as heavy to update. Finding an item may take more time as the system scales.

2.2.2.3 Replication

For fault tolerance and to guarantee the persistence of data[21], each item in the system is replicated across N storage nodes in most distributed storage systems.

The number of individual replicas N, controls a trade-off between performance and fault tolerance. With higher values of N, each data item is replicated across more nodes, so the system becomes more tolerant to individual node failures. With lower values of N, the system becomes more performant, as write operations have to be propagated to less machines. This trade-off has been explored in the past[12], leading to the proposal of multiple replication protocols, such as quorum systems[25] among others [1, 23].

To determine how to distribute the load of N replicas across all nodes, one can use the following techniques:

Master/Slave Replication In master/slave replication, each master node can have multiple slaves. The role of the master node is to receive the updates and replicate the acquired data to all of the saves that it is connected to. The slave nodes are used to answers read calls.

Publish/Subscribe The publish/subscribe replication is characterized by listeners subscribing to channels, with publishers sending data to these channels that connect to subscribers. In the particular case of replication, nodes with an interest on a specific source of data (another node) will subscribe to his channel, receiving the updates from it.

Neighbor Replication Neighbor Replication keeps copies of each item in the $N-1$ neighbors of the node responsible for that key. While this allows to keep tight control on the replication degree, by triggering the creation of new replicas when neighbors change, it has a high maintenance cost, since replicas must be created and destroyed with every change in the network.

Virtual Servers In a system working with Neighbor Replication, one can use Virtual Servers to improve load balancing of the system.

Using this approach, each physical node presents itself as multiple distinct identities to the system. Each identity represents a virtual node maintained by that server. On the other hand, this may amplify the effect of churn[2], since the departure of a single node leads to the simultaneous failure of multiple virtual nodes.

Multi-Publication Multi-Publication stores N replicas of each data item in different and deterministically correlated positions. While this offers very good load balancing properties, it requires a monitoring scheme to detect departure/failure of nodes.

Resilient Load-Balancing This policy uses groups to manage the load balancing of the systems. Each group has a collection of data that is replicated between group nodes. When a new node enters the system, it joins the group with fewest members. Load balancing of data is kept by splitting and merging groups when it reaches a certain unbalanced threshold.

[2]Churn is the participant turnover in the network (the amount of nodes joining and leaving the system per unit of time)

Most Available Node In this policy, data is placed in the nodes predicted to be the most available in the future with higher probability. Using this technique, fewer items are affected by failures and fewer replicas need to be created again. This introduces savings in data transfer costs, but it creates unbalanced load in the system nodes.

2.2.2.4 Multi-version tracking

Some systems that allow weaker consistency, other than serializability, resort to versioning, where multiple versions of the same data item are maintained at the same time. To distinguish between these, there must exist an unique identifier for each version of an object.

Vector Clocks Each update to a data item is always tagged by a vector clock that uniquely identifies that data item version.

A vector clock is a list of pairs (*node*,*counter*), where *node* is the identifier of a node in the system that issued the event/operation being tagged and *counter* is a monotonically increasing value associated with that node.

With this we can keep separate versions of one data item and we can check if two different versions conflict (meaning that how vector clocks encode concurrent events or divergent versions). To do so, we compare the two vector clocks and verify if either one descends from the other or that there is a conflict caused by two separate updates from the same version(i.e. versions have divergent states of an object).

2.2.2.5 Version Reconciliation and Conflict Resolution

When concurrent (and not coordinated) updates are issued over two replicas of a given data object, these replicas will potentially evolve to divergent states, that at some point must be resolved into a single converged state, ideally are that the effects of all operations that generated the divergences. To take this action there are mainly two aspects to consider: how to do it and when to do it. Multiple techniques address this issue, being the main ones briefly described below:

Last Writer Wins With this simple approach the last update based on some notion of time is the one that is adopted by the system. While this is trivial to implement, using local machine clocks, they are usually not synchronized across nodes, which can lead to incorrect decisions and to lost updates.

Programmatic Merge This technique leverages on the programmer to specify what happens when two versions conflict. Every node is required to either have a piece of code that decides how to merge divergent versions of an object or to show both states to the end user on-line and delegate on the user the decision concerning the final converged state.

Commutative Operations

Another approach is to design the system by only allowing commutative operations to be performed, this means that despite the order in which operations are executed, the final result will be the same, as long as all replicas execute all operations. If we can make every update commutative, then conflict resolution becomes only a matter of ensuring that all replicas executed all operations.

A simple example is a counter that only allows increment operations. Independently of the order of these increments, the final value will be the same across all replicas.

Operation Transformation The goal of this technique is to transform operations issued over the system in an automatic fashion as to ensure that they can be applied in any order and still ensure the correctness of the system. This is specially useful when working on collaborative editing.

CRDTs Convergent or Commutative Replicated Data Types. These are distributed data types and structures with mathematical properties that ensure eventual consistency and state convergence while being very scalable and fault-tolerant.

A CRDT[24] requires no synchronization during the execution of operations to ensure state convergence, an update executes immediately, unaffected by network latency, faults, or network failures. There are two kinds of CRDTs:

CvRDTs, State-based Convergent Replicated Data Types. The successive states of an object should form a monotonic semi-lattice and the replica merge operation computes a least upper bound. In other words, the merging of different states will eventually converge to the same final state at each replica. It allows for weak channel assumptions (i.e. unreliable communication channel). However, sending over the entire state might be inefficient for large objects.

CmRDTs, Operation-based Convergent Replicated Data Types. In the operation-based CRDTs, concurrent operations commute. Operation-based replication requires reliable communication channel with ordering guarantees, such as a causal order between operations.

Both classes of CRDTs are guaranteed (by design) to eventually converge towards a common single state.

CRDTs tend to become inefficient over time, as metadata accumulate and internal data structures can become unbalanced[24]. Garbage collection can be performed using some form of weak synchronization, outside of the critical path of client-level operations.

2.2.2.6 Membership/Failure Handling

It is important in distributed storage systems to keep track of which nodes join and leave the system, so we can maintain the guaranties of replication and correct mapping of data objects to nodes. There are several techniques to achieve this, in particular:

Gossip Protocol This technique relies on periodic communication between pairs of nodes to synchronize their local information concerning system membership.

When a node joins the system, it communicates with a set of existing nodes and these periodically contact random peers, synchronizing their membership list. This keeps going until every node in the system knows the existence of the new node. Similar procedure goes for a leaving node.

Anti Entropy Anti Entropy mechanisms are useful to recover from scenarios where nodes fail. These mechanisms detect inconsistencies between replicas and are used to synchronize data between them. This is a specialized form of gossip protocol.

A fast way to detect inconsistencies and minimize the amount of transferred data is to use Merkle Trees[27]. Merkle Tree is a hash tree where leaves are hashes of its keys and parent nodes are hashes of the values of their children. To check if two replicas are synchronized, we compare the hash value at the roots, if it is the same value they are synchronized, if not, the values of the hashes at each level of the tree are recursively compared until all divergent nodes are located. Synchronization is then performed only over the nodes whose values have diverged.

Sloppy Quorum Usually writes need to be performed in a sub-set of nodes, in order to provide consistency and safe persistence. In scenarios where a temporary failure of one of those nodes happens, such operations could not be executed.

Sloppy Quorum allows to write the updated data item to another node, other then the ones previously selected. This node is now responsible to periodically try to contact the original node and deliver the updates that it had previously missed.

Accrual Failure Detection [13] The idea of an Accrual Failure Detection is that the failure detection module does not emit a boolean value stating that node is up or down. Instead, the failure detection module emits a value which represents a suspicion level for each of monitored nodes. The basic idea is to express the value of Φ on a scale that is dynamically adjusted to reflect network and load conditions at the monitored nodes.

2.2.3 Storage System examples

In this section, we will present some of the distributed storage systems that are important for this thesis:

Dynamo [11] Dynamo is a distributed storage system used by several services at Amazon, it uses the key-value storage model, the interface supports get and put operations and it guarantees eventual consistency across data replicas.

It uses an always writable policy for updates, with conflict reconciliation on reads. To do so, data is partitioned and replicated using consistent hashing where the hash values are mapped into node identifiers, that are organized in a circular ring formed by their identifiers, this is done by overlapping the highest hash value next to the lowest one.

Replication of each data item is configurable with the number of replicas N. So besides being stored in the responsible node (according to their hash value) each data object is replicated using neighbor replication. Load balancing is achieved by using virtual servers mapped to the same physical node. It is also possible to adjust the number of nodes that are used to serve read or write operations using parameters R and W.

To keep track of multiple versions of the same item, Dynamo uses vector clocks for version tracking and executes version reconciliation when read operations are submitted on an object. This technique favors write performance.

To handle temporary node failures Dynamo uses a sloppy quorum. Upon the need to recover from a failed node, it uses an anti entropy mechanism backed by Merkle trees to calculate differences between replicas.

Membership is maintained in a distributed way using a gossip protocol.

Redis [7] Redis is a non-relational in-memory database that was built to fill the needs of modern web applications, so its focus is mostly on read performance and availability.

It provides eventual consistency of data while supporting partitioning and replication with publish/subscribe and master/slave techniques. Besides the common key-value interface, it also offers Data structures such as strings, lists, sets, sorted sets, and hashes. Hashes are key-value pairs inside a key and are optimal for storing attributes inside an object.

Cassandra [17] Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers.

Cassandra uses tables similarly to Bigtable. Table columns can have colums inside themselves. To access these, Cassandra offers an API with `insert`, `get`, and `delete` methods.

To distribute data across the system nodes, it uses consistent hashing with a ring layout, similar to Dynamo. Replication is handled with three different policies. "Rack Unaware" replicates each item using neighbor replication, "Rack Aware" and "Datacenter Aware" use a system node as a leader to coordinate the ranges of the replicas

and spreads the copies of each object across machines that are either on different racks on the same datacenter or in different datacenters. To achieve load balancing it uses resilient load-balancing, based on the analysis of the load information of the ring.

Membership is handled with an anti-entropy gossip protocol and it further relies on accrual failure detection for failure detection and handling.

Riak [16] Riak is a distributed key-value data store system that provides high availability by allowing to chose between strong and eventual consistency. Eventual consistency in Riak uses CRDTs at its core, providing CRDT data types that include counters, sets, flags, registers, and maps.

It uses many of Dynamo's concepts, such as consistent hashing for partitioning data around the replicas ring, neighbor replication with virtual nodes to guarantee data availability, vector clocks for object versioning, a gossip protocol to communicate membership changes and an anti-entropy mechanism based on Merkle Trees to detect differences between replicas.

Besides the common interface methods GET, PUT and, DELETE, it also offers Map-reduce querying, secondary indexes, and full-text search functionalities.

Antidote [28] Antidote is a distributed CRDT key-value store written using the Riak Core module that is intended to provide partitioning, Intra-DC (Data center) and Inter-DC replication, while also supporting atomic write transactions.

Antidote has direct support for read and writing operations over CRDTs, this makes it easier for the programmer to reason about and work with these data structures.

The system architecture is divided in the following layers: (i) across DCs replication layer, is responsible for replicating updates to other DC and include both components need in the sender and the receiver DC. The update propagation is done in a FIFO (First in first out) order; (ii) transaction layer, is responsible for providing transaction support, Snapshot Isolation and causality tracking between objects; (iii) materializer, is in charge of forming the objects from the set of operations returned by the logging layer; (iv) logging layer, is used to provide fast write operations, as it appends an operation to the log before the actual update propagation; (v) replication within a DC, uses strict quorum to perform log reads and writes among N replicas within the DC.

This layered architecture design allows to easily add or remove features to the system as well as rely on different strategies for each of these modules.

2.2.4 Distributed Caching Systems

Traditional caching is about using the main memory to temporarily store the most frequently accessed data, in order to speed up te process of fetching data. Caching in a

network context has been used for a long time with the main purpose of improving web access times and reducing traffic and load to the storage to the storage service. With a distributed caching architecture one can achieve better hit ratio, response times, and load balancing[22].

Distributed Caching is becoming more popular since main memory prices have been lowering. Hence, caching is becoming a more relevant layer of the storage service of web applications. Here are some examples of Distributed Caching Systems:

Memcached [29] Memcached is a high-performance, distributed memory object caching system originally intended for use in speeding up dynamic web applications by alleviating the database load.

In Memcached, instead of having a dedicated amount of memory for each node, the whole available memory of the distributed caching service is shared by all the nodes.

Redis Redis can also be used as a caching system if configured to only use the main memory, not dumping periodically to disc.

2.3 Peer-to-Peer Systems

The term "peer-to-peer" (P2P) refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized manner[20].

Peer-to-peer is an alternative to the client-server model, in its purest form there is no notion of server, as every node is a peer that can act as server or client when needed. There are certain characteristics that can be tuned in peer-to-peer systems to achieve a desired working environment. A key aspect of peer-to-peer systems is overlay networks, logical networks that operate at the application level and that are used to manipulate and control the interactions among the participants of the systems.

2.3.1 Degree of Decentralization

Decentralized

In a fully decentralized system, every peer is an equal participant. This way we can avoid bottlenecks and single points of failure, while promoting scalability and resilience. On the other hand, it is hard to keep a global view of such a network, since there is no coordinator node. Usually these networks rely on flooding protocols to propagate changes.

To address this issue, some systems elect supernodes in order to balance load in the network and becoming an entry point for new participants.

Partly Centralized

Partly Centralized systems have a centralized component that stores information

about the available resources and acts as a coordinator to the other nodes. This component is responsible for managing node connections.

These systems are simpler to build than fully decentralized ones, but have a potential of bottleneck and single point of failure in the centralized component, where unavailability can potentially render the system impossible to access.

2.3.2 Structured vs Unstructured

This design option is usually based on how useful it is to have a performant exact match querying mechanism and the amount of churn in the network.

Structured Network

Structured networks are specially useful to make efficient queries in the network. Each node has a unique identifier that is used to discover its location among the remaining nodes. Most structured overlay networks rely on consistent hashing to operate, forming a ring array with nodes of the system accordingly to their identifiers, behaving as a distributed hash table (DHT). This structure works similarly to a traditional hash table, where a value can be easily found by its key.

Building and maintaining such structure has costs. The DHT needs to constantly update the list of active nodes, which can be very expensive if network churn is high.

Unstructured Network

In an unstructured network each node keeps its own collection of resources that surround it and updates are propagated using flooding mechanisms. Not having a particular structure makes the network more tolerant to churn, but propagating queries becomes a heavier task as the topology preaches no indication of the location of resources, leading the query to be propagated through a gossip like protocol to all the other participants of the system.

2.4 Distributed Coordination

In large scale distributed applications, there is a need for some sort of coordination mechanism that suits specifically the application's working environment.

Coordination in distributed systems helps tackle problems such as[10] (i) event ordering, (ii) mutual exclusion, (iii) atomicity, (iv) concurrency control, (v) deadlock handling, (vi) leader election, (vii) reaching an agreement.

2.4.1 Distributed Coordination Services

ZooKeeper [14] ZooKeeper is a service for coordinating processes of distributed applications. This service exposes an API as a coordination kernel that provides the

necessary tools for the developer to create coordination mechanisms. In order to show the developer how to use the API to create high level coordination primitives, these mechanisms are then integrated in a package of coordination recipes. These allow to build primitives such as (i) configuration management, allowing to implement dynamic configuration in distributed applications; (ii) rendezvous, this is useful to share information between a master process and several workers; (iii) group membership, in order to keep track of the operational nodes in the system; (iv) locks, as a method to modify a certain variable in mutual exclusion mode. Besides this, there can also be implemented read and write locks.

ZooKeeper provides to it's clients an abstraction of a set of data nodes that are organized by hierarchical name spaces. This data structure can be manipulated by the clients in order to implement the coordination mechanisms mentioned before.

2.5 Data Serialization

In a distributed system messages are constantly being sent and received between several nodes. This communication between nodes takes a crucial role in the system performance[26], so most systems today use data structure serialization mechanisms. These mechanisms turn the application's internal objects into a structured format that can be sent to another node.

These formats usually support a way to manipulate the written data, so it can be easily created by the sending node and easily read and parsed by the receiving node.

2.5.1 Data Serialization mechanisms

XML [6] XML is a serialization mechanism that is expressed as a markup language. XML was developed by members of the W3C and has been around for a while, thus creating a large user base around the world.

XML was developed having in mind a straightforwardly use over the internet, so it is widely supported. The message's format allows to be parsed without knowing it's schema in advance and it is human readable.

Despite it's easy usability and support, one of it's drawbacks is the very verbose syntax.

JSON [5] JSON, JavaScript Object Notation is a lightweight format to serialize data. JSON expresses the content of a javascript object and so it can represent four primitive data types (string, numbers, booleans and null) and two structured types (objects and arrays).

It is widely used in a browser context, and as well as XML, it is human readable, but it removes the overhead of the verbose markup.

However, JSON also has it's shortcomings, such as extensibility drawbacks and lack of input validation.

Protocol Buffers [30] Protocol buffers is the google's solution for data serialization. Oppositely to XML and JSON, it encodes data into a binary format, so it can have a smaller size going through the network. Also, the message format are defined in a .proto file, that will be used to serialize and parse the content of the message. Without the knowledge of the it's format it is not trivial to decode the message.

The message format specification is done a syntax similar to JSON. The field types can be numbers (integer or floating-point), booleans, strings, raw bytes, or even other protocol buffer message.

This binary format suits the needs of modern distributed applications by reducing the size of the messages traded between nodes. A drawback of using a binary serialization is not being human readable.

2.6 Summary

This chapter has covered the existing work that will support, influence, and help the development of this thesis.

The main focus is around storage systems and the many techniques they use to tackle different challenges in the design and operation of those distributed systems. The major aspects surround consistency, partitioning, replication, object/update versioning, version reconciliation and membership/failure handling.

This chapter also discussed peer-to-peer systems and some aspects of overlay networks, since they are widely used in many distributed systems and in particular to implement storage systems.

Besides that, the fundamentals of WebRTC, a crucial aspect for the work done on the Legion framework that is also one of the goals of the work to be developed in this thesis, was briefly introduced.

In the next chapter, we discuss the planned work for this thesis, with its major goals and time planning.

Chapter 3

Integration Design and Architecture

This chapter presents the design of the solution developed in this work to connect Legion clients to Antidote data storage. We start by briefly presenting Antidote and Legion, focusing on the mechanisms necessary for integrating both systems. After this introduction, we presents an overview of the proposed design and the interaction between Legion and Antidote.

3.1 Overview of used systems

In order to design an integration between Legion and Antidote, it is imperative that we understand how both systems work, so we can take advantage of their internal mechanisms to our advantage. Next follows a brief presentation of the mentioned systems, as well as some important working details.

3.1.1 Legion

A large number of web application are built around direct interactions among clients, such as collaborative applications, social networks and multi-user games. These applications manage a set of shared objects, with each user reading and writing to a subset of these objects. These kind of applications are usually implemented using a centralized infrastructure that maintains the shared state and mediates all interactions among users. This approach has some drawbacks, such as the servers becoming a scalability bottleneck, the service interrupt when the servers are unavailable and the latency of nearby users being unnecessarily high.

One alternative to this model, is to rely on direct communication between users. In a not so far past, such an alternative would be troublesome when combining peer-to-peer

and Web applications. Due to the lack of supporting mechanisms for this, it was often needed browser extensions or plugins to implement such a scenario.

With the recent development of tools such as WebRTC, that enables direct communication between browsers and connectivity utilities such as STUN and TURN that can overcome the trouble of connecting users between NATs and firewalls, a new set of peer-to-peer web applications started to be developed.

Legion is a framework that exploits these new features for enriching web applications with direct data replication between browsers. Each client maintains a local data storage with replicas of a subset of application shared objects. Legion uses an eventual consistency model, where each client can modify its local objects without coordination, this allows updates to be performed concurrently on different replicas while modifications are propagated asynchronously. To guarantee that all replicas converge to the same correct state after updates have been applied on different replicas, Legion relies of CRDTs as its internal data structure. To support these interactions in a peer-to-peer manner, clients form overlay networks to propagate objects and updates among them. In each overlay network, a few clients act as synchronization point of a possible centralized component. These clients have an objects server component that is design to support extensions with legacy storage systems to act as centralized persistent storage. This component has a view of the Legion group objects, which can be used as a replication point of the system state to a centralized storage system. The external system will use Legion's primitives, available in the communication API, in order to propagate updates back and forward. These updates are sent in a point-to-multipoint fashion, by delegating the afford of propagating the updates to all the legion clients in the group to a node which carries the objects server. Overall, image 3.1 illustrates how an external centralized storage system can interact with Legion, as well as Legion's architecture.

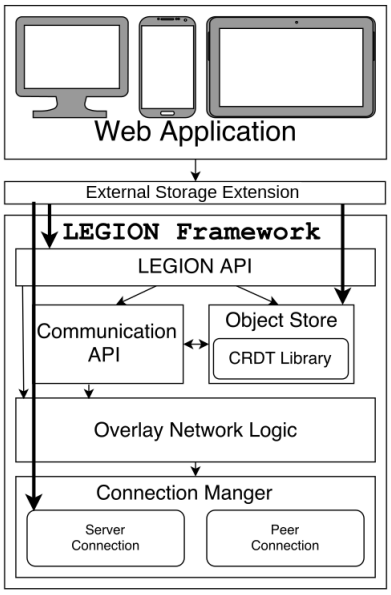

Figure 3.1: Legion Architecture

3.1.2 Antidote

Traditional databases provide strong guarantees but are slow and unavailable under failures and network partition. Hence, they are not suitable for geo-replication. The alternatives are NoSQL-style databases which are fast and available even under network partition. They provide a low-level key-value interface and expose data inconsistencies due to asynchronous communication among the servers. It takes significant effort and expertise from programmers to deal with these inconsistencies and develop correct applications on top of these databases. Antidote provides features that aid programmers to write correct applications, while having the same performance and horizontal scalability as NoSQL, from a single machine to geo-replicated deployments, with the added guarantees of Causal Highly-Available Transactions, and provable absence of data corruption due to concurrency.

Internally, Antidote uses CRDTs as data structures. To modify these, one can make use of interactive transaction composed by three steps: start, update/read and finally commit. Transactions can receive a timestamp to force the read/update to be executed at a certain system snapshot. Another import API method is the ability to request the executed operations log since a certain timestamp. Antidote clients can use this API via the distributed Erlang interface that can only be used by Erlang clients, or the protocol buffer interface, which is language independent. Antidote can work in cluster mode, with several nodes in the same cluster that replicate data using the saved logs, or in inter-dc mode by simply replicating data in a FIFO (First in first out) operation order. The system architecture can be visualized in image 3.2.

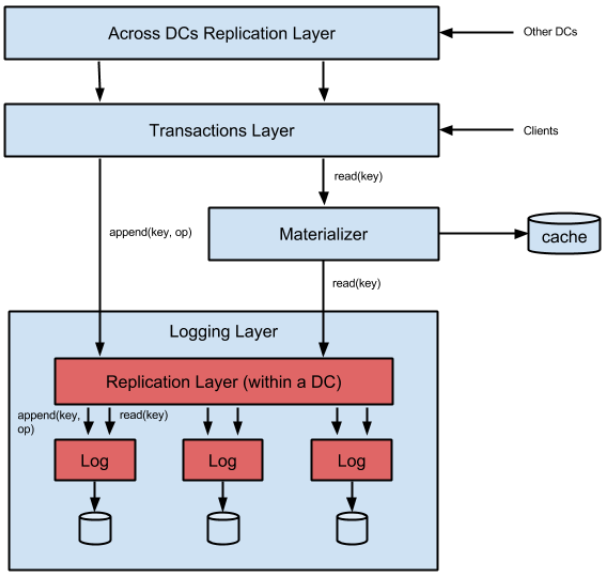

Figure 3.2: Antidote Architecture

3.2 Architecture Overview

Legion includes a mechanism to allow the integration with other external storage systems. This mechanism consists in a module that implements a pre-defined interface.

Figure 3.3 shows the architecture of our approach to integrate Legion with Antidote. The following components are included in the architecture.

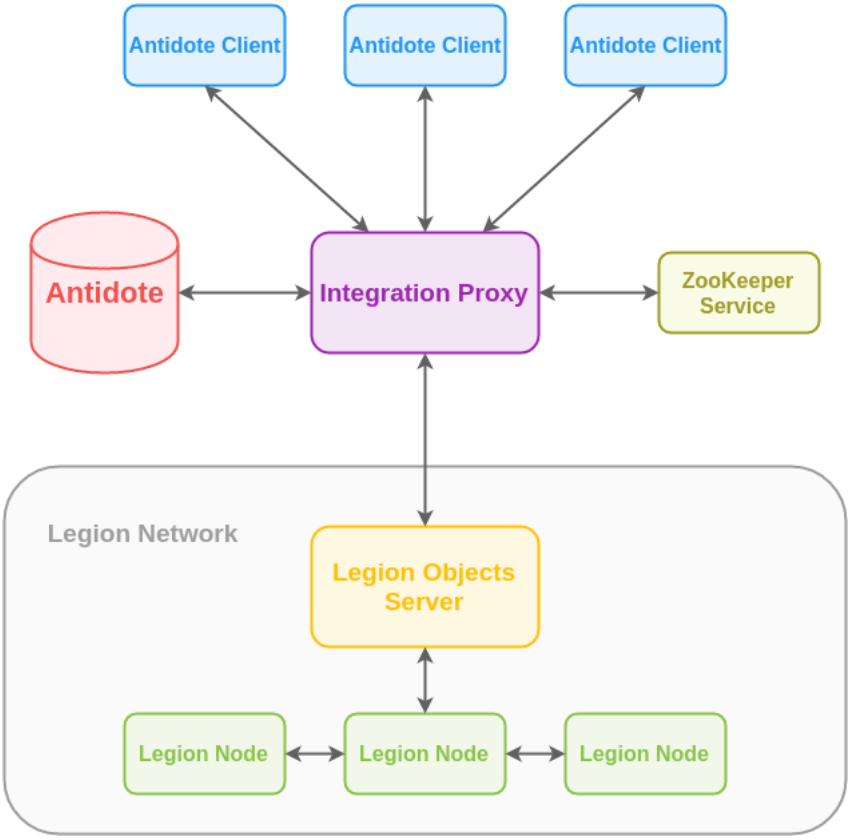

Figure 3.3: Integration Architecture

The integration of Legion and Antidote can be depicted into the following components:

Antidote

The storage system to be integrated with Legion. Antidote will act as a persistent, stable, client-server model storage system. It will be synchronized with Legion, so both will have the entire set of updated objects in the system. An Antidote node can have it's data updated directly from an antidote client, or from an update that was propagated from Legion nodes.

Besides that, Antidote will also keep meta-data related to the system synchronization, such as propagated operations identifiers and elements' unique tokens.

Antidote Client

This component is one of the entry points of information into the system. An

Antidote client can manipulate the system's data set by making calls to modify the state of Antidote's objects.

This client is written in javascript and it uses the integration proxy in order to make read and write requests because a direct request can not be made from a browser due to cross-origin security. These requests have a flag to distinguish them from the objects server requests.

Legion Node

This component is a Legion instance in the system. It acts as an entry point of information to the system and it also as the responsibility to keep it self and the other Legion nodes updated and synchronized.

Legion's Objects Server

This component keeps a global view of what is happening in the Legion network. There can be one or more objects servers in the system, and Legion's synchronization events will pass through at least one of these. In the objects server also resides the logic for the synchronization between Antidote and Legion. This is done by watching for object update events and then propagating them.

This component is written in NodeJS and it uses the integration proxy for the communication with Antidote, by making read and write calls.

Integration Proxy

This component is needed to establish communication with Antidote. There can be one or more integration proxies in the system. It takes advantage of Antidote's protocol buffer interface to make read and write calls.

The proxy can receive requests from either a Antidote client or a Legion's object server synchronization method. The request is parsed and a call is sent to Antidote. When the answer from antidote is received, it is redirected to who made the request. This component is written in NodeJS.

ZooKeeper Service

ZooKeeper is used in the system as a service in order to guarantee mutual exclusion of meta-data updates from different sources. It receives locking requests from all the integration proxies and keeps information of the locking status of each system object.

3.3 Integration challenges

When integrating both systems, there were some challenges there we had to tackle and overcome. The main effort in this integration is the synchronization, initially we tried to follow the path of merging CRDTs directly, but as explained before, Antidote didn't offer the needed mechanisms to make this happen. The synchronization was then changed

to be about operation propagation. This brings some questions regarding the needed metadata and the operations order. For the first topic, it was needed to keep operation identifiers, as well as CRDT tokens, both stored in Antidote. Having metadata modified from different sources heeds concerns about data corruption. We need to assure that updates to this structure is done safely. About the second topic, applying past operations in another systems requires that the updates are executed at a certain system snapshot.

The solution of these challenges are depicted next in sections 3.4 and 3.5, by explaining step by step how the data is propagated between systems, keeping the system's objects synchronized along with the related metadata.

3.4 Legion to Antidote flow

This section explains the process of propagating an update issued by a Legion node into Antidote. Algorithm 1 depicts the sequence of events.

In order to send the updates from Legion to Antidote we must first detect these updates. To do so, we opted for an event driven model, by relying on Legion's objects server, since every update issued by a Legion node will end up in at least one of the objects servers.

Once the event is detected in the objects server, the incoming message is filtered based on the event type. If the event contained in the message is indeed an update, that objects server will issue one or more update requests to the integration proxy. Once a request reaches the integration proxy, this one will decouple the request into the the actual update data and meta-data. These two are then sent to Antidote as write requests. Besides writing the data into Antidote, the Integration proxy certifies that the operation has not been executed already, since there can be more than one objects server to issue the request. This is done by checking the meta-data stored in Antidote.

Once the update reaches Antidote successfully, the integration proxy returns to the objects server.

Algorithm 1 Legion to Antidote flow

```
1:  Upon event:
2:      if event.type = update then
3:          objectsServer.apply(update)
4:          metaData ← parse(update)
5:          doneOperation ← antidote.read(doneOperations)
6:          if !(update.id in doneOperations) then
7:              zooKeeper.lock(metadata.objectId)
8:              antidote.write(metadata)
9:              zooKeeper.unlock(metadata.objectId)
10:             antidote.write(update.operations, lastSeenTimestamp)
11:             return
```

3.5 Antidote to Legion flow

This section explains the process of propagating an update issued by an Antidote client into Legion. Algorithm 2 shows this process step by step.

When an Antidote client issues an update, it is firstly sent to the integration Proxy. This one receives the request and sends them to Antidote. This is due to the cross-origin security. The integration proxy returns to the client when the update is successful.

The next step is to send this update to every Legion node. This can be done by sending the update to the objects servers, which will then propagate the message to every node.

At this point there are two methods of sending the update to the objects server. It can be done with a public/subscribe system that fires an event each time an Antidote object is updated, or having Legion's objects servers to probe periodically for new updates on Antidote. The first option would definitely be more performant, because scanning Antidote periodically for updates would have some overhead. With this in mind we talked to the development team in order to know the best way to to this. One alternative was to use Antidote's internal updates propagation mechanism to also fire an event to the exterior, but this solution had two major problems. The update propagation mechanism would not handle lost messages, this would have to be handled by us, and to exposed such internal mechanism to the outside would go against the Antidote's design path. With this, we concluded that we should proceed to the second option.

In order to periodically check for updates in Antidote, we used the system log to get the operations executed since a certain timestamp.

Legion's objects servers will periodically make a request to the integration proxy, in order to check the operations executed in Antidote since the last seen timestamp. The proxy relays the request to antidote and returns the result. If there are no new updates, then the objects server only updates it's last seen timestamp. If there are new updates that were not propagated to legion, the objects server executes the operation locally, and then propagates it to every Legion node.

Algorithm 2 Antidote to Legion flow

```
1:  Every 2 seconds:
2:      logOperations ← antidote.get(logOperations, lastSeenTimeStamp)
3:      for each operation ∈ logOperations do
4:          if !(operation.id in objectsServer.doneOperations) then
5:              metadata ← parse(operation)
6:              zooKeeper.lock(metadata.objectId)
7:              antidote.write(metadata)
8:              zooKeeper.unlock(metadata.objectId)
9:              objectsServer.Apply(operation)
10:             legion.Propagate(operation)
11:     objectsServer.update(lastSeenTimestamp)
12:     return
```

Chapter 4

Integration Implementation

In this chapter we will focus on the implementation of the integration, describing the work done in each component, the connection between components and emphasizing some important implementation details.

To cover this chapter we will start by (i) describing the supported data types and how they translate between systems, then we will (ii) enumerate the changes we deemed needed to Antidote, which are mainly an afford to make the system more language abstract. After this, we will show as well the (iii) changes made to Legion, in the objects server. The next component depicted is the (iv) integration proxy, which was developed from scratch in order to communicate with Antidote. After the component's description we will get a more in depth description of the update propagation flow, from (v) Legion to Antidote, and then from (vi) Antidote to Legion.

4.1 Supported Data Types

This section describes the supported data types from Legion and how they translate to the correspondent Antidote data types.

4.1.1 Sets

This integration supports sets as one of the CRDTs that Legion offers. More precisely, the supported sets in this integration are operation sets.

Legion holds it's data types in the objectStore component and internally implements this operation set as a OR-Set, as described in [24]. An ORSet is a CmRDT, which characterizes it self as an operation based CRDT that propagates it's changes based on the operations done. This suits our model of operation propagation.

In order to store this data type in Antidote, we naturally also chose an OR-Set, as Antidote supports it natively.

When an operation is done in a system's OR-Set, the same operation will be issued to the other system's OR-Set. This process is straight forward, but we need to keep track of the unique identifiers that pair with each element of the set. This is important, for example, when deleting an element from the set. We want to delete a particular element with a unique id, not any element that matches. Because of this, there are also metadata sets in Antidote that store this correspondence between Legion and Antidote unique tokens.

4.1.2 Counters

One of the supported data types in Legion are Counters. They are internally implemented as a state based PN-Counter, as described in [24]. Using a CvRDT means that the object will converge by propagating and merging its state. In order to match this data type in Antidote, we used an operation based PN-Counter.

To make the two counters synchronized, we must execute increments and decrements based on who issued the update. When a counter is updated in Legion, since this is a state based PN-Counter, its internal state will be propagated to at least one object server. With the counter's state in our hands, we will make a diff based on the current state of the Antidote's counter, the changes are then applied, in order to get the same state on both. When updating a counter in Antidote, since this one is operation based, we can simply apply the increments and decrements directly in the object server. In both cases we have to keep a meta data set to store the correspondence between operation IDs. This is necessary to avoid duplicated operations when more than one object server catches the same update.

4.2 Antidote changes

In this section we will describe the changes to Antidote that were needed in order to make this integration possible.

Antidote is a storage system based in Riak Core and it is still in a fresh stage of development. It is written in Erlang and it offers two interfaces for clients to communicate with: (i) distributed Erlang interface, and (ii) protocol buffer interface.

The first one uses distributed Erlang and can only be used by an Erlang client, which is of no use to us in the integration.

The second one uses google's protocol buffer serialization mechanism to offer an interface that can be language abstract. This would suit our needs since we want to be able to communicate with Antidote via the integration proxy written in NodeJS. Although we were able to communicate with Antidote like this, we soon discovered that this interface was using Erlang binaries over the protocol buffers, which were extremely difficult to

encode/ decode in the NodeJS side. Besides this, it was missing some important calls comparing to the distributed Erlang interface.

In order to proceed with the integration the following features needed to be available in the protocol buffer interface:

- protocol buffer messages could only contain basic data types, like numbers or strings for the interface to be language abstract.
- API methods that returned not only the object's value, but also it's hidden information, like commit timestamps and unique identifiers.
- An API method that returns the content of the system log since a certain timestamp.

With the help from Antidote's development team, these requirements were implemented. The missing interface methods were already implemented internally, so it was only required to make an interface method to export those to the exterior.

In order to make the interface language abstract, it was implemented a JSON data structure over protocol buffer messages. This was possible by having a JSON encoding/ decoding component in Antidote's protocol buffer interface. Although this might not be the best solution, since it partially defeats the purpose of protocol buffers, it was the more practical solution regarding this thesis time span.

Summing up, in listing 4.1 we show the protocol buffer interface methods before this change, and listing 4.2 shows the only method that these were encapsulated into, where the 'value' field is a JSON object with the request.

After these changes were implemented, we were able to proceed with the system integration. Algorithm 3 shows how Antidote interacts with the other components. When handling a write request, Antidote propagates the update internally and replies with information from that transaction. When the request is a read, Antidote replies with the object itself, including unique identifiers and transaction information. The requests are always paired with a timestamp that stands for the last moment that the system was synchronized. The internal object update propagation can be intra-datacenter or inter-datacenter, depending on the current system setup.

Algorithm 3 Antidote request interaction

```
1: Upon request:
2:     Switch request.type
3:         case "write"
4:             answer ← updateObjects(request.objectId, request.timestamp)
5:             Send reply(answer)
6:         case "read"
7:             answer ← getObjects(request.objectId,request.timestamp)
8:             Send reply(answer)
```

```
message ApbReadObjects {
  repeated ApbBoundObject boundobjects = 1;
  required bytes transaction_descriptor = 2;
}

message ApbGetObjects {
  repeated ApbBoundObject boundobjects = 2;
}

message ApbUpdateObjects {
  repeated ApbUpdateOp updates = 1;
  required bytes transaction_descriptor = 2;
}

message ApbStartTransaction {
  required ApbVectorclock timestamp = 1;
  optional ApbTxnProperties properties = 2;
}

message ApbCommitTransaction {
  required bytes transaction_descriptor = 1;
}

message ApbGetLogOperations {
  repeated ApbVectorclock timestamps = 2;
  repeated ApbBoundObject boundobjects = 3;
}
```

Listing 4.1: Protocol Buffer interface methods before

```
message ApbJsonRequest {
  required bytes value = 1;
}
```

Listing 4.2: Protocol Buffer interface methods after

4.3 Legion changes

The majority of the code from the implementation was developed in Legion, more specifically in the objects server component. This is where the logic of the synchronization resides.

To manage the synchronization between Legion and Antidote there two main parts that needed to be implemented: (i) propagate updates that reach the objects server from a legion node, and (ii) periodically check for updates that were issued directly to Antidote.

4.3.1 Legion to Antidote flow

As mentioned before in 3.4, every update issued by a Legion node will be caught in the objects server. Algorithm 4 shows the steps needed for the objects server to propagate each message. A message that reaches the objects server can be of several types, but the ones that interests us are the ones having the type 'contentFromNetwork'. By catching this event we know that an update has been issued, so now we can parse the message and get the details that we need in order to propagate the operation to Antidote. Listing 4.3 shows an example of an incoming update message.

After parsing the message, a request is sent to the integration proxy, containing the element, the operation and the unique identifiers needed. The remaining process is then handled by the integration proxy, that returns when the update propagation was successful.

Algorithm 4 Objects Server Legion to Antidote update propagation

```
1: Upon message:
2:     if message.type = "contentFromNetwork" then
3:         operationInfo ← parse(message.content)
4:         Send request(operationInfo)
5:         Upon answer:
6:             return answer
```

```
{
  type:"OS:C",
  sender:1,
  ID:7012120591,
  content:{
    type:"OPLIST",
    objectID:"objectID2",
    operations:[
      {
        dependencyVV:{},
        opID:1,
        result:{
          element:"a",
          unique:"6651411189259640"
        },
        opName:"add",
        clientID:"1"
      }
    ]
  },
  destination:"localhost:8004"
}
```

Listing 4.3: Legion update content message example

4.3.2 Antidote to Legion flow

As mentioned before in 3.5, the process of propagating updates from Antidote to the Legion network is based on periodic probing.

Algorithm 5 shows the pseudo-code of the objects server work in propagating updates from Antidote to Legion. In order to periodically check for new updates in Antidote, there is a method that starts running when both the objects server and Antidote start. This method fires an event every two seconds that makes a request to the integration proxy for the getLogOperations method. This request contains the element to search for and the vector clock that matches the last seen timestamp of Antidote's state in the objects server. This method returns an object with the list of updates that were executed in Antidote since the given timestamp until the current time. An example of the operations log response is shown in listing 4.4.

If there exists new updates to the element, these will be locally executed in the objects server, and then propagated to the all the Legion nodes. After this, another request is made to the integration proxy, in order to keep the track of the correspondent unique identifiers from Antidote to Legion.

At the end of this process we also need to update the last seen timestamp of Antidote's state in the objects server, this information also comes in the answer from the getLogOperations request. On a side note, the time interval between probes to the Antidote log should be tuned to best fit the current setup.

Algorithm 5 Objects Server Antidote to Legion update propagation

1: **Every** 2 seconds:
2: $updateList \leftarrow$ getLogOperations($elements, lastSeenTimestamp$)
3: **for each** $update \in updateList$ **do**
4: $identifiers \leftarrow$ execute($update$)
5: propagate($update$)
6: Send request($upxdate.Id, identifiers$)
7: $lastSeenTimestamp \leftarrow updateList.timestamp$

```
{log_operations:[
  {
    clocksi_payload:{
        key: {json_value:"objectID2"}
    },
    type:"crdt_orset",
    update:{
      add:[
        {json_value:"b"},
        {binary64:"YySEUYb9uvr4EMO3glI8BWTKurw="}
      ]
    },
       commit_time:[
         {dcid:["antidote@127.0.0.1",1472,53665,947954]},
```

```
15          1472053805871290
16      ]
17    }
18 ]}
```

Listing 4.4: Antidote log response example

4.4 Proxy Development

This component, as mentioned in 3.2, links Antidote to the other components that need to exchange data.

The integration proxy as two main objectives in this system: (i) work as a proxy by relaying the requests from both Antidote clients and Legion's objects servers to Antidote, and (ii) handle the meta-data stored in Antidote, by always keeping it updated. Algorithm 6 depict the logic of the work done by the integration proxy.

In order to receive requests from the above mentioned components, this proxy acts as an HTTP server that exposes an interface with API routes to communicate with Antidote. The available API is shown in listing 4.5. The API methods are simple, the methods are the same for all data types, being this resolved by the route in case of a GET or in the request payload in case of a PUT.

After this, the proxy will need to send a request of it's own to Antidote. In order to do so, we make use of the protocol buffer interface mentioned in 4.2.

The second goal of the proxy is to keep the system's meta-data updated in Antidote. To attain this, every time an update is requested to the proxy, this one has to make a request to Antidote in order to update the object, and another request to update the meta-data object. When handling an object's meta-data, there can be data corruption when more then one update is done concurrently over the same object, so in order to avoid this, we used the ZooKeeper system to act as a distributed locking mechanism that makes each update request to acquire a lock. This guarantees that we have mutual exclusion per object when manipulating the meta-data objects.

Every Antidote object as a related meta-data object. For example, if we have a set called 'objectID2', then there is a meta-data object called 'objectID2_tokens' that contains a set of pairs that represents the correspondence between Antidote and Legion unique identifiers.

```
1 server.route({
2   method: 'GET',
3   path: '/ping',
4   handler: function (request, reply) {
5     ...
6   }
7 }
8
9 server.route({
```

```
10    method: 'GET',
11    path: '/readObjects/{key}/{type}',
12    handler: function (request, reply) {
13      ...
14    }
15  }
16
17  server.route({
18    method: 'GET',
19    path: '/getObjects/{key}/{type}',
20    handler: function (request, reply) {
21      ...
22    }
23  }
24
25  server.route({
26    method: 'PUT',
27    path: '/updateObjects',
28    handler: function (request, reply) {
29      ...
30    }
31  }
32
33  server.route({
34    method: 'GET',
35    path: '/getLogOps/{key}/{type}/{vClock}',
36    handler: function (request, reply) {
37      ...
38    }
39  }
```

Listing 4.5: Integration proxy API

Algorithm 6 Integration Proxy request handling

```
1:  Upon request:
2:      Switch request.type
3:          case "write"
4:              data, metadata ← parse(request)
5:              lock(metadata)
6:              Send request(metadata)
7:              Upon answer:
8:                  unlock(metadata)
9:              Send request(data)
10:             Upon answer:
11:                 Send reply(answer)
12:         case "read"
13:             data ← parse(request)
14:             Send request(data)
15:             Upon answer:
16:                 Send reply(answer)
```

Chapter 5

Evaluation

In this chapter we present the evaluation of the work done. This will measure the performance and scalability of the integration by presenting benchmarks of the running system.

Section 5.1 will measure how the number of operation each message contains influences the time needed to propagate them. In section 5.2 will be shown how the meta-data stored in Antidote increases with the operations executed in the system.

5.1 Operation Propagation Time

In this section we will focus on measuring the propagation time of system operations and how it fluctuates with other variables. In this test scenario, we use one Antidote instance, one legion object server instance, one Antidote client and one Legion node. All the system components are located in the same network. This experiment consists in saving the current time in one client, when the update is issued. The update is sent and when the other client detects to have received the update, saves the time and finally makes the difference between the two times.

Firstly, in figure 5.1 we measure the operation propagation time scaling with the number of operations in each message. As we can see, the time it takes to propagate a message, increases with the number of operations that each message contains. When propagating messages from Legion to Antidote, the time required increases nearly linearly with the number of operations. On the other hand, when propagating messages from Antidote to legion, the time required does not grow as fast, only increasing slightly with the number of operations in each message.

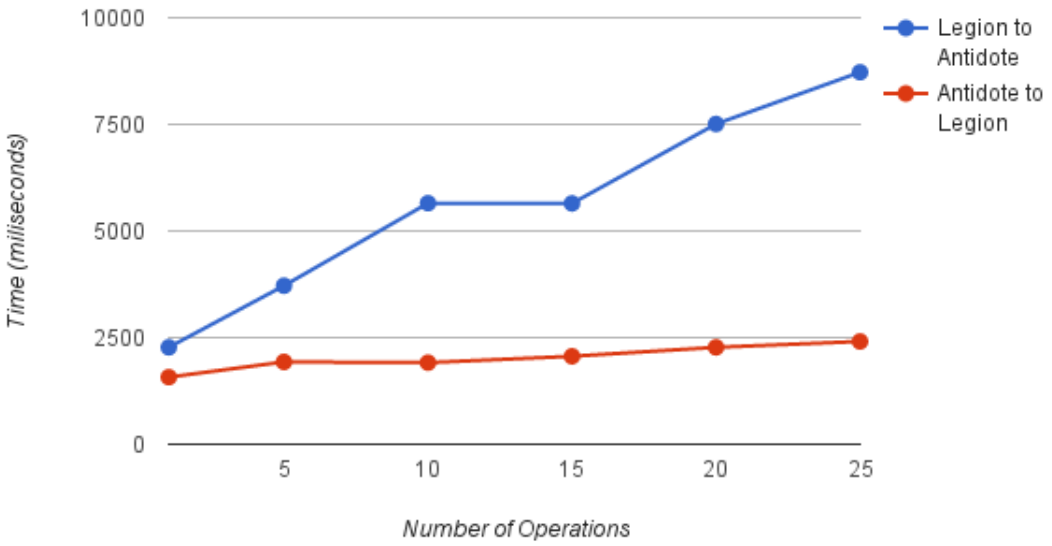

Figure 5.1: Propagation time scaling with operations per message

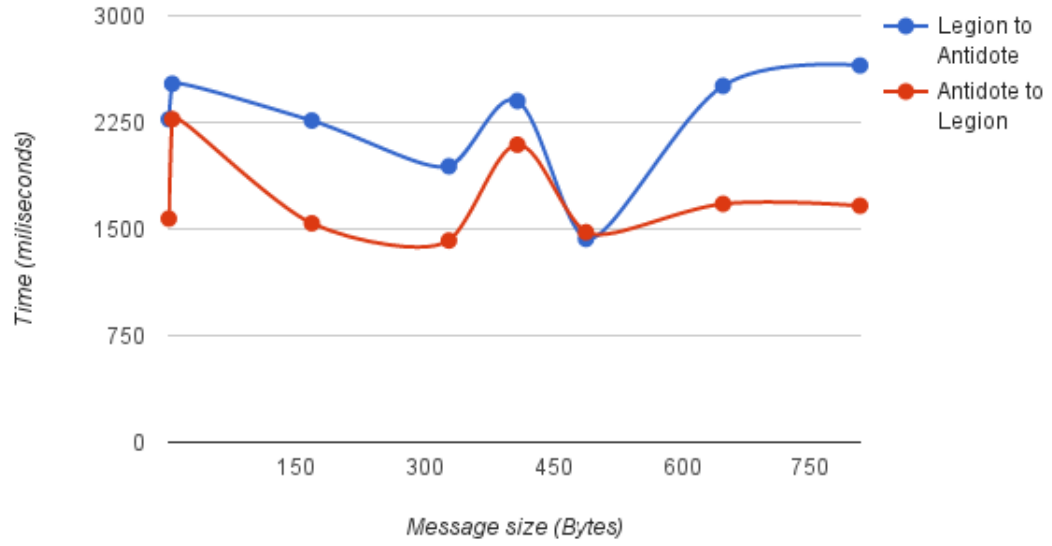

Figure 5.2: Propagation time scaling with message size

Secondly, in 5.2 we measure the scalability of the message propagation time with the size of the message. As we can see, the time needed to propagate a message does not seem to scale with the size of the message. Although the updates from Antidote to Legion still propagate faster, both update directions are not influenced by the size of the message,

resulting in a chart with fluctuating time values and no growing pattern.

In the next evaluation scenario, we will measure the scalability of propagating data between two separate Legion groups that can only share information via Antidote. In order to do so, this evaluation setup is composed by one Antidote instance, two Legion object server instances and two Legion nodes connected to separate object servers. In this scenario, the time measured starts when a legion node issues an update and ends when the other group's Legion node receives the update. In order to propagate updates between Legion groups, Antidote is used as a synchronization point, where one Legion node updates the storage system, and the other fetches the updates. In image 5.3 we measure the time of propagating messages between the two Legion nodes scaling with the number of operations in each message. As we can see, the time needed scales nearly linearly with the number of operations, having a consistent result with the charts shown before.

In image 5.4 we measure the scalability of the message propagation time with the size of each message. Similarly to chart 5.2, it does not seem to exist a scaling pattern with the increasing message size.

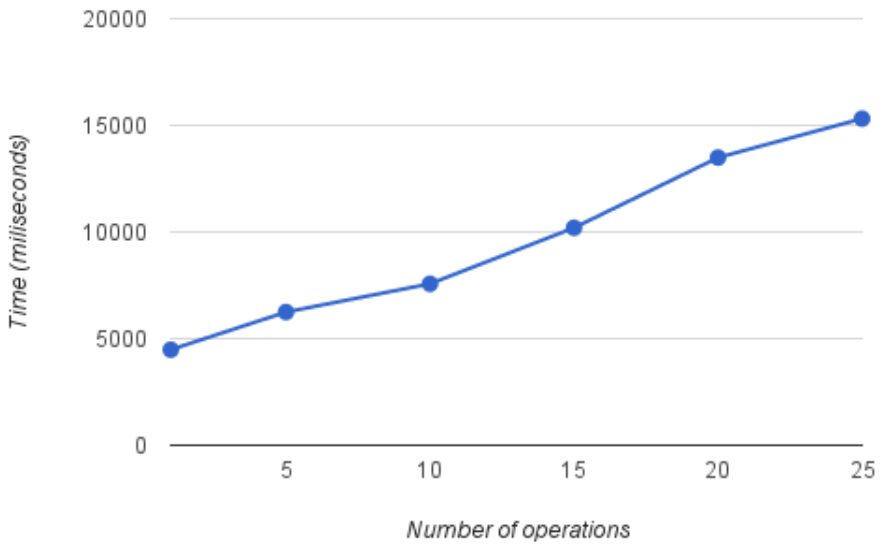

Figure 5.3: Propagation time scaling with operations per message between two Legion groups

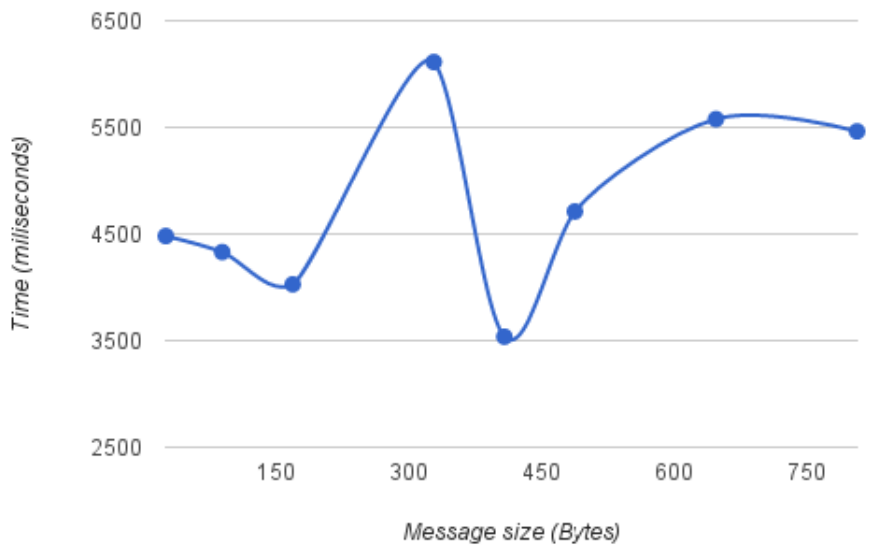

Figure 5.4: Propagation time scaling with message size between two Legion groups

5.2 Meta-Data Size

As explained in 4.4, in order to keep the system from propagating duplicated operations, there is a need to store meta-data information in the Antidote nodes. In this section we measure the size of this meta-data along with it's scalability as the system runs. The measured values in figure 5.5 are a representation of the size of meta-data stored in an Antidote node. In this test scenario, we consider write and delete operations, as these separately will influence the final size of meta-data. The blue line represents a workload where 50% of the updates are writes and the other 50% are deletes. On the other hand, the red line has a more write intensive workload, with 80% writes and 20% deletes.

As expected, the meta-data size stored increases with the number of operations. In the red workload, the growing is continuously linear. The blue workload also grows continuously, but needs less memory. This is due to the unique tokens that are erased when a delete operation is issued, making the only stored data the operation identifiers.

The always growing tendency of the meta-data derives from the operation identifiers that will keep stacking as the system runs. This points the lack of a garbage collector in order to clean the no longer needed operation identifiers from time to time.

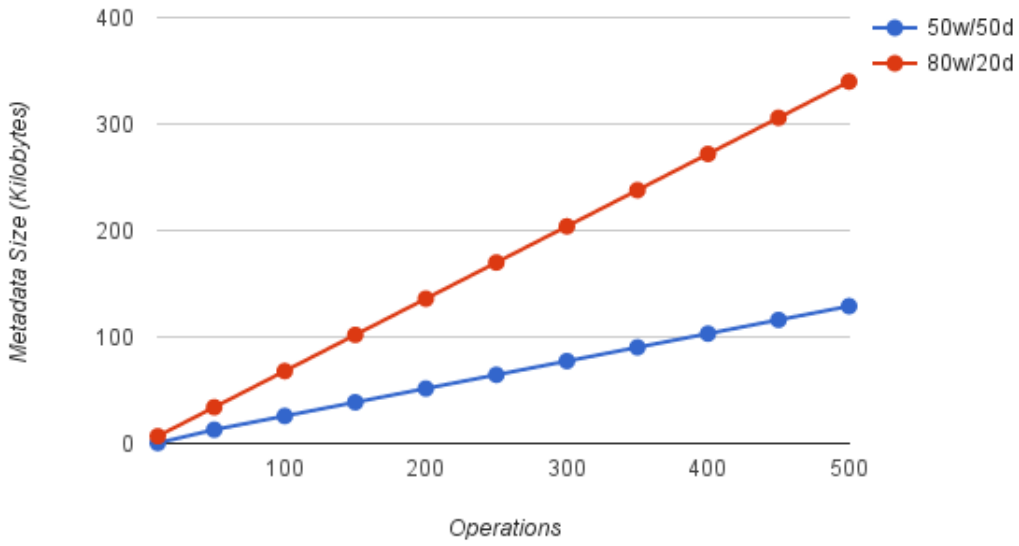

Figure 5.5: Meta-data size scaling with propagated operations

Chapter 6

Conclusion

The work done in this thesis centers around Legion, a framework that uses peer-to-peer communication between clients in order to achieve improvements in client-to-client latency, lowering bandwidth usage on the server and supported disconnected operations from the centralized component.

One of the possible development paths for legion is the persistence integration with legacy storage systems, in order to offer a safe and stable storage solution and to allow users running new and old application to interact. This would make possible for client-server applications to run alongside Legion leveraged applications. Starting from this point, this thesis' work focuses in integrating Legion with Antidote, in order to achieve one of the goals for the Legion project.

We designed and implemented this integration, so that an Antidote based application can run alongside an application that uses Legion. In order to keep the two systems synchronized, we used an operation propagation logic, being reactive to events from Legion to Antidote and pro-active probing from Antidote to Legion.

To validate the implementation, we conducted a stack of benchmarks tests that can show us some metrics related to the integration performance and scalability. This metrics are based on message propagation time, stored meta-data size and variables such as number of clients, number of operations and operation message size.

In summary, the main contributions of the work presented in this thesis are as follows:

- The development of a more language abstract interface for Antidote.
- The implementation of a javascript client for Antidote.
- The design and implementation of the integration between Legion and Antidote.
- An evaluation of the integration implementation.

6.1 Technical decisions

In the starting point of this thesis, the main objective was to integrate Legion with two data storage systems that also used CRDTs, Riak and Antidote. Riak is more enterprise oriented, with several real world use cases, while Antidote is still very fresh and under development. Even thought Antidote uses Riak Core, the two systems diverge a lot based on design options and it's objectives. We started by speaking to a few developers from each of the systems, being able to conclude that Antidote's development environment would be much more beneficial for us regarding support. This was a critical matter since no one directly involved in this thesis was an expert in either of the systems, neither we had much experience in the programming language used (Erlang). With this in mind, we followed to start the integration with Antidote.

Our first integration design thought was to use the CRDT's data structures of both systems, translate one to another, and then merge them in order to propagate the state of the system. Once again we talked to the development team as to get a guideline of how this could be done. As we understood, it was not the intended design path to expose the internal CRDT's data structure to the outside of the system. The system's CRDTs were meant to only be accessed as an object abstraction.

After acknowledging this, the design of the integration was changed to be based on propagating operations, by executing on one system the operation that was made on the other.

6.2 Future Work

After this thesis development, in order to consolidate and complement this integration, a number of future improvements can be mentioned.

The data types available in this implementation are sufficient to demonstrate a working application that uses both these systems, but in a real world scenario, there would be the need to have more data types in order to correctly support the creation of different application types.

The current Antidote protocol buffer interface allows us send and receive JSON objects serialized as protocol buffers 'bytes' data type. As this suffices for a working interface, it is not how protocol buffers were intended to be used. Another upgrade, not only to this integration, but also to the Antidote project, would be a refactoring of Antidote's protocol buffer interface.

Bibliography

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. "Exploiting atomic broadcast in replicated databases". In: *Euro-Par'97 Parallel Processing*. Springer, 1997, pp. 496–503.

[2] S. Almeida, J. Leitão, and L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 85–98.

[3] P. Bailis and A. Ghodsi. "Eventual Consistency Today: Limitations, Extensions, and Beyond". In: *Commun. ACM* 56.5 (May 2013), pp. 55–63. ISSN: 0001-0782. DOI: 10.1145/2447976.2447992. URL: http://doi.acm.org/10.1145/2447976.2447992.

[4] A. Bergkvist, D Burnett, and C. Jennings. "A. Narayanan," WebRTC 1.0: Real-time Communication Between Browsers". In: *World Wide Web Consortium WD WD-webrtc-20120821* (2012).

[5] T. Bray. "The javascript object notation (json) data interchange format". In: (2014).

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. "Extensible markup language (XML)". In: *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210* 16 (1998), p. 16.

[7] J. L. Carlson. *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013. ISBN: 1617290858, 9781617290855.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[9] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[10] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[12] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

[13] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. "The phi; accrual failure detector". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. 2004, pp. 66–78. DOI: 10.1109/RELDIS.2004.1353004.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX Annual Technical Conference*. Vol. 8. 2010, p. 9.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663.

[16] R. Klophaus. "Riak Core: Building Distributed Applications Without Shared State". In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 14:1–14:1. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900176. URL: http://doi.acm.org/10.1145/1900160.1900176.

[17] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[18] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[19] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño Martínez, and J. E. Armendáriz-Iñigo. "Snapshot Isolation and Integrity Constraints in Replicated Databases". In: *ACM Trans. Database Syst.* 34.2 (July 2009), 11:1–11:49. ISSN: 0362-5915. DOI: 10.1145/1538909.1538913. URL: http://doi.acm.org/10.1145/1538909.1538913.

[20] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. *Peer-to-peer computing*. 2002.

[21] J. Paiva and L. Rodrigues. "Policies for Efficient Data Replication in P2P Systems". In: *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. 2013, pp. 404–411. DOI: 10.1109/ICPADS.2013.63.

[22] S Paul and Z Fei. "Distributed caching with centralized control". In: *Computer Communications* 24.2 (2001), pp. 256 –268. ISSN: 0140-3664. DOI: http://dx.doi.org/10.1016/S0140-3664(00)00322-4. URL: http://www.sciencedirect.com/science/article/pii/S0140366400003224.

[23] F Schenider. "Replication management using the state-machine approach, Distributed Systems". In: *Ed. Sape Mullender*, (1993), pp. 169–198.

[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[25] D. Skeen. *A quorum-based commit protocol*. Tech. rep. Cornell University, 1982.

[26] A. Sumaray and S. K. Makki. "A comparison of data serialization formats for optimal efficiency on a mobile platform". In: *Proceedings of the 6th international conference on ubiquitous information management and communication*. ACM. 2012, p. 48.

[27] M. Szydlo. "Merkle tree traversal in log space and time". In: *Advances in Cryptology-EUROCRYPT 2004*. Springer. 2004, pp. 541–554.

Webography

[28] *Antidote README*. Accessed: 2016-02-01. URL: https://github.com/SyncFree/antidote.

[29] *Memcached About*. Accessed: 2016-01-26. URL: http://memcached.org/about.

[30] *Protocol buffers: Google's data interchange format*. Accessed: 2016-08-19. URL: https://developers.google.com/protocol-buffers/docs/overview.

[31] *WebRTC in the real world: STUN, TURN and signaling*. Accessed: 2016-02-08. URL: http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/.